Chapter SIXTEEN
Stream Operations on Collections

Exam Objectives

Develop code that uses Stream data methods and calculation methods.
Use java.util.Comparator and java.lang.Comparable interfaces.
Sort a collection using Stream API.

Comparator and Comparable

To sort arrays or collections, Java provides two very similar interfaces:

- **java.util.Comparator**

```
public interface Comparator<T> {
    int compare(T obj1, T obj2);

    // Other default and static methods ...
}
```

- **java.lang.Comparable**

```
public interface Comparable<T> {
    int compareTo(T obj);
}
```

The difference is that `java.util.Comparator` is implemented by a class you use to sort **ANOTHER** class' objects while `java.lang.Comparable` is implemented by the **SAME** object you want to sort.

With `Comparator`, you make an object to compare two objects of another type to sort them; that's why you take two parameters, and the method is called compare (those two objects).

With `Comparable`, you make an object comparable to another object of the same type to sort them, that's why you only take **ONE** parameter and the method is called `compareTo` (that other object). Since this interface is easier to grasp, let's start with it.

java.lang.Comparable

The method to implement is:

```
int compareTo(T obj);
```

As you can see, it returns an int. Here are its rules:

- When **ZERO** is returned, it means that the object is **EQUAL** to the argument.
- When a number **GREATER** than zero is returned, it means that the object is **GREATER** than the argument.
- When a number **LESS** than zero is returned, it means that the object is **LESS** than the argument.

Many classes of Java (like `BigDecimal`, `BigInteger`, wrappers like `Integer`, `String`, etc.) implement this interface with a natural order (like `1`, `2`, `3`, `4` or `A`, `B`, `C`, `a`, `b`, `c`).

Since this method can be used to test if an object is equal to another one, it's recommended that the implementation is consistent with the `equals(Object)` method (if the `compareTo` method returns `0`, the equals method must return `true`).

Once a object implements this interface, it can be sorted by `Collections.sort()` or `Arrays.sort()`. Also, it can be used as key in a sorted map (like `TreeMap`) or in a sorted set (like `TreeSet`).

The following is an example of how an object can implement `Comparable`.

```
public class Computer implements Comparable<Computer> {
    private String brand;
    private int id;

    public Computer(String brand, int id) {
        this.brand = brand;
        this.id = id;
    }

    // Let's compare first by brand and then by id
    public int compareTo(Computer other) {
        // Reusing the implementation of String
        int result = this.brand.compareTo(other.brand);

        // If the objects are equal, compare by id
        if(result == 0) {
            // Let's do the comparison "manually"
            // instead of using Integer.compareTo()
            if(this.id > other.id) result = 1;
            else if( this.id < other.id) result = -1;
            // else result = 0;
        }
        return result;
    }

    // equals and compareTo must be consistent
    // to avoid errors in some cases
    public boolean equals(Object other) {
        if (this == other) return true;
        if (!(other instanceof Computer)) return false;
        return this.brand.equals(other)
            && this.id == ((Computer)other).id;
    }

    public static void main(String[] args) {
        Computer c1 = new Computer("Lenovo", 1);
        Computer c2 = new Computer("Apple", 2);
        Computer c3 = new Computer("Dell", 3);
        Computer c4 = new Computer("Lenovo", 2);

        // Some comparisons
        System.out.println(c1.compareTo(c1)); // c1 == c1
        System.out.println(c1.compareTo(c2)); // c1 > c2
```

```
        System.out.println(c2.compareTo(c1)); // c2 < c1
        System.out.println(c1.compareTo(c4)); // c1 < c2
        System.out.println(c1.equals(c4)); // c1 != c2

        // Creating a list and sorting it
        List<Computer> list = Arrays.asList(c1, c2, c3, c4);
        Collections.sort(list);
        list.forEach(
           c -> System.out.format("%s-%d\n",c.brand,c.id)
        );
    }
}
```

When you execute this program, this is the output:

```
0
11
-11
-1
false
Apple-2
Dell-3
Lenovo-1
Lenovo-2
```

java.util.Comparator

The method to implement is:

```
int compare(T o1, T o2);
```

The rules of the returned value are similar than `Comparable` 's:

- When **ZERO** is returned, it means that the **FIRST** argument is **EQUAL** to the **SECOND** argument.
- When a number **GREATER** than zero is returned, it means that the **FIRST** argument is **GREATER** than the **SECOND** argument.
- When a number **LESS** than zero is returned, it means that the **FIRST** argument is **LESS** than the **SECOND** argument.

One advantage of using a `Comparator` instead of `Comparable` is that you can have many `Comparator` s to sort the same object in different ways.

For instance, we can take the `Computer` class of the previous example to create a `Comparator` that sorts first by id and then by brand, and since the rules of the returned value are practically the same as `Comparable` 's, we can use the `compareTo` method:

```
Comparator<Computer> sortById =
    new Comparator<Computer>() {
        public int compare(Computer c1, Computer c2) {
            int result = Integer.compare(c1.id, c2.id);
            return result == 0
                ? c1.brand.compareTo(c2.brand) : result;
    }
};
```

Also, `Integer.compare(x, y)` is equivalent to:

```
Integer.valueOf(x).compareTo(Integer.valueOf(y))
```

Luckily, `Comparator` is a functional interface, so we can use a lambda expression instead of an inner class:

```
Comparator<Computer> sortById = (c1, c2) -> {
    int result = Integer.compare(c1.id, c2.id);
    return result == 0
       ? c1.brand.compareTo(c2.brand) : result;
}
```

So, when we use it in the list of the previous example:

```
List<Computer> list = Arrays.asList(c1, c2, c3, c4);
Collections.sort(list, sortById);
list.forEach(
   c -> System.out.format("%d-%s\n",c.id,c.brand)
);
```

The output is:

```
1-Lenovo
2-Apple
2-Lenovo
3-Dell
```

In case you're wondering, `Comparable` is also considered a functional interface, but since `Comparable` is expected to be implemented by the object being compared, you'll almost never use it as a lambda expression.

In Java 8, with the introduction of default and static methods in interfaces, we have some useful methods on `Comparator` to simplify our code like:

```
Comparator<T>
   Comparator.comparing(Function<? super T, ? extends U>)
Comparator<T>
   Comparator.comparingInt(ToIntFunction<? super T>)
Comparator<T>
   Comparator.comparingLong(ToLongFunction<? super T>)
Comparator<T>
   Comparator.comparingDouble(ToDoubleFunction<? super T>)
```

That takes a `Function` (a lambda expression) that returns the value of a property of the object that will be used to create a `Comparator` using the value returned by `comparedTo` (also notice the versions when you're working with primitives).

For example:

```
Comparator<Computer> sortById =
          Comparator.comparing(c -> c.id);
```

Or:

```
Comparator<Computer> sortById =
          Comparator.comparingInt(c -> c.id);
```

They are equivalent to:

```
Comparator<Computer> sortById = new Comparator<Computer>() {
    public int compare(Computer c1, Computer c2) {
        return Integer.valueOf(c1.id)
```

```
                    .compareTo(Integer.valueOf(c2.id));
    }
};
```

Another useful method is `thenComparing` that chains two `Comparators` (notice that this is not a `static` method):

```
Comparator<T>
    thenComparing(Comparator<? super T>)
Comparator<T>
    thenComparing(Function<? super T, ? extends U>)
Comparator<T>
    thenComparingInt(ToIntFunction<? super T>)
Comparator<T>
    thenComparingLong(ToLongFunction<? super T>)
Comparator<T>
    thenComparingDouble(ToDoubleFunction<? super T>)
```

This way, we can simplify the code to create a `Comparator` to sort by id and then by brand by using:

```
Comparator<Computer> sortByIdThenByBrand =
        Comparator.comparing((Computer c) -> c.id)
            .thenComparing(c -> c.brand);
```

Finally, the default method `reverse()` will create a `Comparator` that reverses the order of the original `Comparator` :

```
List<Computer> list = Arrays.asList(c1, c2, c3, c4);
Collections.sort(list,
    Comparator.comparing((Computer c) -> c.id).reversed());
list.forEach(
    c -> System.out.format("%d-%s\n",c.id,c.brand));
```

The output:

```
3-Dell
2-Apple
2-Lenovo
1-Lenovo
```

Sorting a Stream

Sorting a stream is simple. The method

```
Stream<T> sorted()
```

Returns a stream with the elements sorted according to their natural order. For example:

```
List<Integer> list = Arrays.asList(57, 38, 37, 54, 2);
list.stream()
    .sorted()
    .forEach(System.out::println);
```

Will print:

```
2
37
38
54
57
```

The only requirement is that the elements of the stream implement `java.lang.Comparable` (that way, they are sorted in natural order). Otherwise, a `ClassCastException` may be thrown.

If we want to sort using a different order, there's a version of this method that takes a `java.util.Comparator` (this version is not available for primitive stream like `IntStream`):

```
Stream<T> sorted(Comparator<? super T> comparator)
```

For example:

```
List<String> strings =
    Arrays.asList("Stream","Operations","on","Collections");
strings.stream()
    .sorted( (s1, s2) -> s2.length() - s1.length() )
    .forEach(System.out::println);
```

Or:

```
List<String> strings =
    Arrays.asList("Stream","Operations","on","Collections");
strings.stream()
    .sorted( Comparator.comparing(
                (String s) -> s.length()).reversed() )
    .forEach(System.out::println);
```

Both will print:

```
Collections
Operations
Stream
on
```

The first snippet of code will return a positive value if the first string length is less than the second's, and a negative value otherwise, to sort the string in descending order.

The second snippet of code will create a `Comparator` with the length of the string in natural order (ascending order) and then reverse that order.

Data and Calculation Methods

The `Stream` interface provides the following data and calculation methods:

```
long count()
Optional<T> max(Comparator<? super T> comparator)
Optional<T> min(Comparator<? super T> comparator)
```

And in the case of the primitive versions of the `Stream` interface, we have the following methods:

IntStream

```
OptionalDouble average()
long count()
OptionalInt max()
OptionalInt min()
int sum()
```

LongStream

```
OptionalDouble average()
long count()
OptionalLong max()
OptionalLong min()
long sum()
```

DoubleStream

```
OptionalDouble average()
long count()
OptionalDobule max()
OptionalDouble min()
double sum()
```

`count()` returns the number of elements in the stream or zero if the stream is empty:

```
List<Integer> list = Arrays.asList(57, 38, 37, 54, 2);
System.out.println(list.stream().count()); // 5
```

`min()` returns the minimum value in the stream wrapped in an `Optional` or an empty one if the stream is empty.

`max()` returns the maximum value in the stream wrapped in an `Optional` or an empty one if the stream is empty.

When we talk about primitives, is easy to know which the minimum or maximum value is. But when we are talking about objects (of any kind), Java needs to know how to compare them to know which one is the maximum and the minimum. That's why the `Stream` interface needs a `Comparator` for `max()` and `min()`:

```
List<String> strings =
    Arrays.asList("Stream","Operations","on","Collections");
strings.stream()
    .min( Comparator.comparing(
                (String s) -> s.length())
    ).ifPresent(System.out::println); // on
```

`sum()` returns the sum of the elements in the stream or zero if the stream is empty:

```
System.out.println(
    IntStream.of(28,4,91,30).sum()
); // 153
```

`average()` returns the average of the elements in the stream wrapped in an `OptionalDouble` or an empty one if the stream is empty:

```
System.out.println(
    IntStream.of(28,4,91,30).average()
); // 38.25
```

Key Points

- `java.util.Comparator` is implemented by a class you use to sort **ANOTHER** class' objects. `java.lang.Comparable` is implemented by the **SAME** object you want to sort.
- The main methods of both interfaces return an `int`. Their rules are very similar:
  - When **ZERO** is returned, it means that the object (or first argument) is **EQUAL** to the (second) argument.
  - When a number **GREATER** than zero is returned, it means that the object (or first argument) is **GREATER** than the (second) argument.
  - When a number **LESS** than zero is returned, it means that the object (or first argument) is **LESS** than the (second) argument.
- `comparing()`, `thenComparing()`, and `reverse()` are helper methods of the `Comparator` interface added in Java 8.
- The `sorted()` method of the `Stream` interface returns a stream with the elements sorted according to its natural order. You can also pass a `Comparator` as an argument.
- `count()` returns the number of elements in the stream or zero if the stream is empty.
- `min()` returns the minimum value in the stream wrapped in an `Optional` or an empty one if the stream is empty.
- `max()` returns the maximum value in the stream wrapped in an `Optional` or an empty one if the stream is empty.
- `sum()` returns the sum of the elements in the stream or zero if the stream is empty.
- `average()` returns the average of the elements in the stream wrapped in a `OptionalDouble` or an empty one if the stream is empty.

Self Test

1. Given:

```
public class Question_16_1 {
    public static void main(String[] args) {
        List<String> strings =
            Arrays.asList( "Stream","Operations","on","Collections");
        Collections.sort(strings, String::compareTo);
        System.out.println(strings.get(0));
    }
}
```

What is the result?
A. `Collections`
B. `on`
C. Compilation fails
D. An exception occurs at runtime

2. Which of the following statements returns a valid `Comparator`?
A. `(String s) -> s.length();`
B. `Comparator.reversed();`
C. `Comparator.thenComparing((String s) -> s.length());`
D. `Comparator.comparing((String s) -> s.length() * -1);`

3. Given:

```
public class Question_16_3 {
    public static void main(String[] args) {
```

```
        List<Integer> list = Arrays.asList(30, 5, 8);
        list.stream().max().get();
    }
}
```

What is the result?
A. `5`
B. `30`
C. Compilation fails
D. An exception occurs at runtime

4. Given:

```
public class Question_16_4 {
    public static void main(String[] args) {
        List<String> strings =
            Arrays.asList( "Stream","Operations","on","Collections");
        strings.stream()
            .sorted(
                Comparator.comparing(
                    (String s1, String s2) ->
                            s1.length() - s2.length()
                )
            )
            .forEach(System.out::print);
    }
}
```

What is the result?
A. `CollectionsOperationsStreamOn`
B. `onStreamOperationsCollections`
C. Compilation fails
D. An exception occurs at runtime

Open answers page

---

15. Data Search

17. Peeking, Mapping, Reducing and Collecting